COMPUTER SCIENCE-PROJECT FILE

MADE BY:

Name: Pranav Verma

Class: XI Raman

Roll No: 8

Session: 2024-25

RSA Cryptography System

A Public Key Cryptography Implementation

Contents

Certificate

This is to certify that **PRANAV VERMA** of class **XI RAMAN** has prepared a **RSA CRYPTOGRAPHY SYSTEM**. The report project is the result of his efforts and endeavours. The report is found worthy of acceptance as the project report under the subject computer science of class XI. He has prepared the report under my guidance.

(Parul Kapil)

Acknowledgement

I would like to express my special thanks of gratitude to my teacher MS. PARUL KAPIL as well as our principal DR. RUCHI SETH who gave me the golden opportunity to do this wonderful project on the topic RSA CRYPTOGRAPHY SYSTEM, which also helped me in doing a lot of Research and I came to know about so many new codes and tags I am really thankful to them. Secondly, I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Chapter 1

Introduction

1.1 RSA Cryptography Algorithm

RSA (Rivest Shamir Adleman) is one of the first public-key algorithms and is widely used for secure data transmission. The encryption key is public and distinct from the decryption key which is kept secret (private). In RSA, this is based on the difficulty of factoring the product of two large prime numbers.
This Project Utilizes a custom-made, simpler version of the RSA Algorithm to allow for **Message Encryption and Decryption**.

1.2 Features

The system includes the following key features:

- This program will generate a public-private key pair, and store it on the user's computer.
- The User will then be able to use that key pair to encrypt and decrpt messages.
- Users also will be able to create signatures, which are essentially verification of the message's sender.
- This Program has a User-Friendly Command Line Interface.

Chapter 2

How This Project Works

2.1 Key Generation Process

The key generation process follows these steps:

1. Choose two distinct prime numbers p and q
2. Find $n = p \times q$
3. Calculate Euler's totient function: $\phi(n) = (p-1)(q-1)$
4. Choose public exponent e where $1 < e < \phi(n)$ and $gcd(e, \phi(n)) = 1$
5. Find private exponent d where $d \equiv e^{-1} \pmod{\phi(n)}$

```
def generate_keys():
    primes = []
    for i in range(100, 501):
        is_prime = True
        for j in range(2, int(math.sqrt(i)) + 1):
            if i % j == 0: #check for prime
                is_prime = False
                break
        if is_prime:
            primes.append(i)

    p = random.choice(primes) # random select prime 1
    q = random.choice(primes) # random select prime 2
    while p == q: # the two numbers should not be equal
        q = random.choice(primes)

    n = p * q # calc. base
    phi = (p - 1) * (q - 1) #eulers totient function

    e = random.randint(2, phi - 1)
    while gcd(e, phi) != 1: # calc gcd
        e = random.randint(2, phi - 1)

    d = pow(e, -1, phi)
```

```
    return ((e, n), (d, n))
```

Listing 2.1: Key Generation Algorithm

2.2 Message Encryption Process

The encryption process converts plaintext into ciphertext using the recipient's public key. For each character in the message:

$$ciphertext = plaintext^e \bmod n \tag{2.1}$$

Where (e, n) is the public key. Here's the implementation:

```
def encryptMessage(message, public_key):
    e, n = public_key
    encrypted_message = []
    for char in message:
        encrypted_char = (ord(char) ** e) % n
        encrypted_message.append(str(encrypted_char))
    return ' '.join(encrypted_message)
```

Listing 2.2: Message Encryption Function

2.3 Message Decryption Process

The decryption process converts ciphertext back to plaintext using the recipient's private key:

$$plaintext = ciphertext^d \bmod n \tag{2.2}$$

Where (d, n) is the private key:

```
def decrypt_message(encrypted_message, private_key):
    d, n = private_key
    encrypted_chars = encrypted_message.split()
    decrypted_message = ''.join(
        [chr((int(char) ** d) % n) for char in encrypted_chars]
    )
    return decrypted_message
```

Listing 2.3: Message Decryption Function

2.4 Digital Signature System

The digital signature system ensures that the message being transmitted is from the original source, and has not been tampered with.

2.4.1 Signature Creation

- Calculate message hash using SHA-256
- Encrypt the hash with sender's private key
- Attach encrypted hash as signature

```
def hash_message(message):
    return hashlib.sha256(message.encode()).hexdigest()

def sign_message(message, private_key):
    d, n = private_key
    message_hash = hash_message(message)
    signature = []
    for char in message_hash:
        # ok how did this actually work
        signed_char = pow(ord(char), d, n)
        signature.append(str(signed_char))
    return ' '.join(signature)
```

Listing 2.4: Signature Creation

2.4.2 Signature Verification

- Decrypt signature using sender's public key
- Calculate hash of received message
- Compare decrypted signature with calculated hash

```
def verify_signature(message, signature, public_key):
    e, n = public_key
    # make the hash from original sig.
    signature_nums = signature.split()
    recovered_hash = ''
    for num in signature_nums:
        char = pow(int(num), e, n)
        recovered_hash += chr(char)
    # compare
    return recovered_hash == hash_message(message)
```

Listing 2.5: Signature Verification

2.5 Combined Encryption and Signing

The Program is able to do both at the same time, to save on time:

```
def encrypt_and_sign(message, recipient_public_key,
                     sender_private_key):
    encrypted_message = encryptMessage(message,
                                       recipient_public_key)
```

```
    signature = sign_message(encrypted_message,
                             sender_private_key)
    return encrypted_message, signature

def verify_and_decrypt(encrypted_message, signature,
                       sender_public_key, recipient_private_key):
    if not verify_signature(encrypted_message, signature,
                            sender_public_key):
        return False, "Signature verification failed!"
    decrypted_message = decrypt_message(encrypted_message,
                                        recipient_private_key)
    return True, decrypted_message
```

Listing 2.6: Encryption with Signature

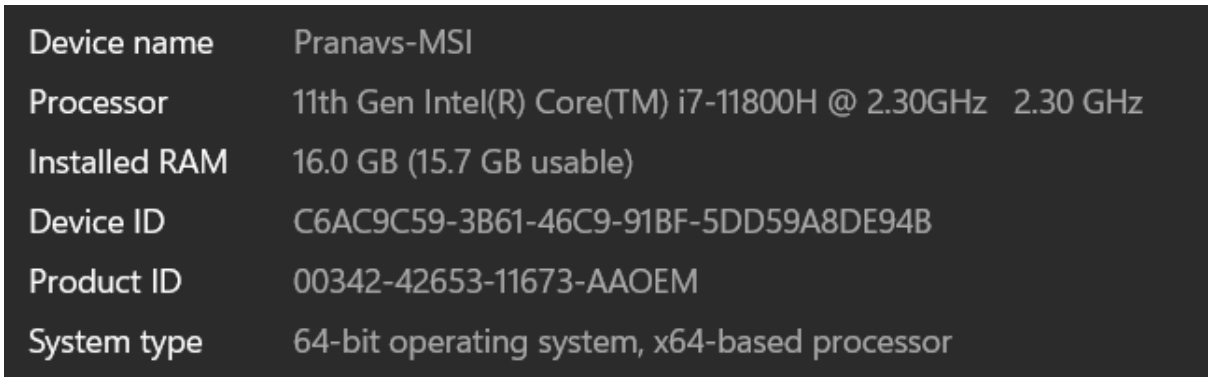
Chapter 3

How To Use This Program

1. User generates key pair or loads existing keys
2. For sending a message:
    - User inputs message
    - Message is encrypted with recipient's public key
    - Encrypted message is signed with sender's private key
    - Both encrypted message and signature are saved to files
3. For receiving a message:
    - System loads encrypted message and signature
    - Verifies signature using sender's public key
    - Decrypts message using recipient's private key
    - Displays decrypted message if verification succeeds

Chapter 4

Hardware Used

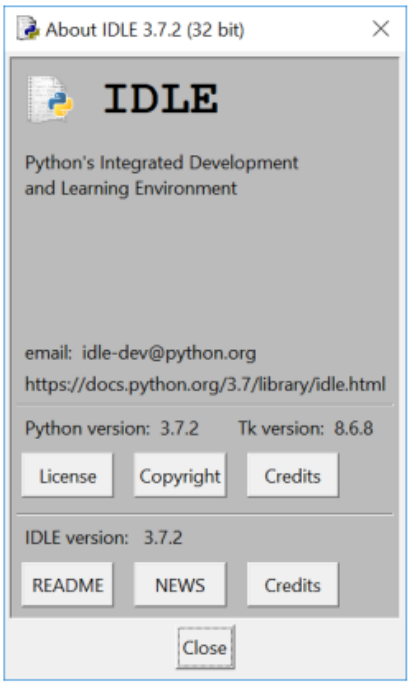


Device name	Pranavs-MSI
Processor	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz 2.30 GHz
Installed RAM	16.0 GB (15.7 GB usable)
Device ID	C6AC9C59-3B61-46C9-91BF-5DD59A8DE94B
Product ID	00342-42653-11673-AAOEM
System type	64-bit operating system, x64-based processor

This PC was used to make this project.

Chapter 5

Software Used



Python 37-32 was used to write the code of this system.

Chapter 6

System Architecture

The main components are organized as follows:

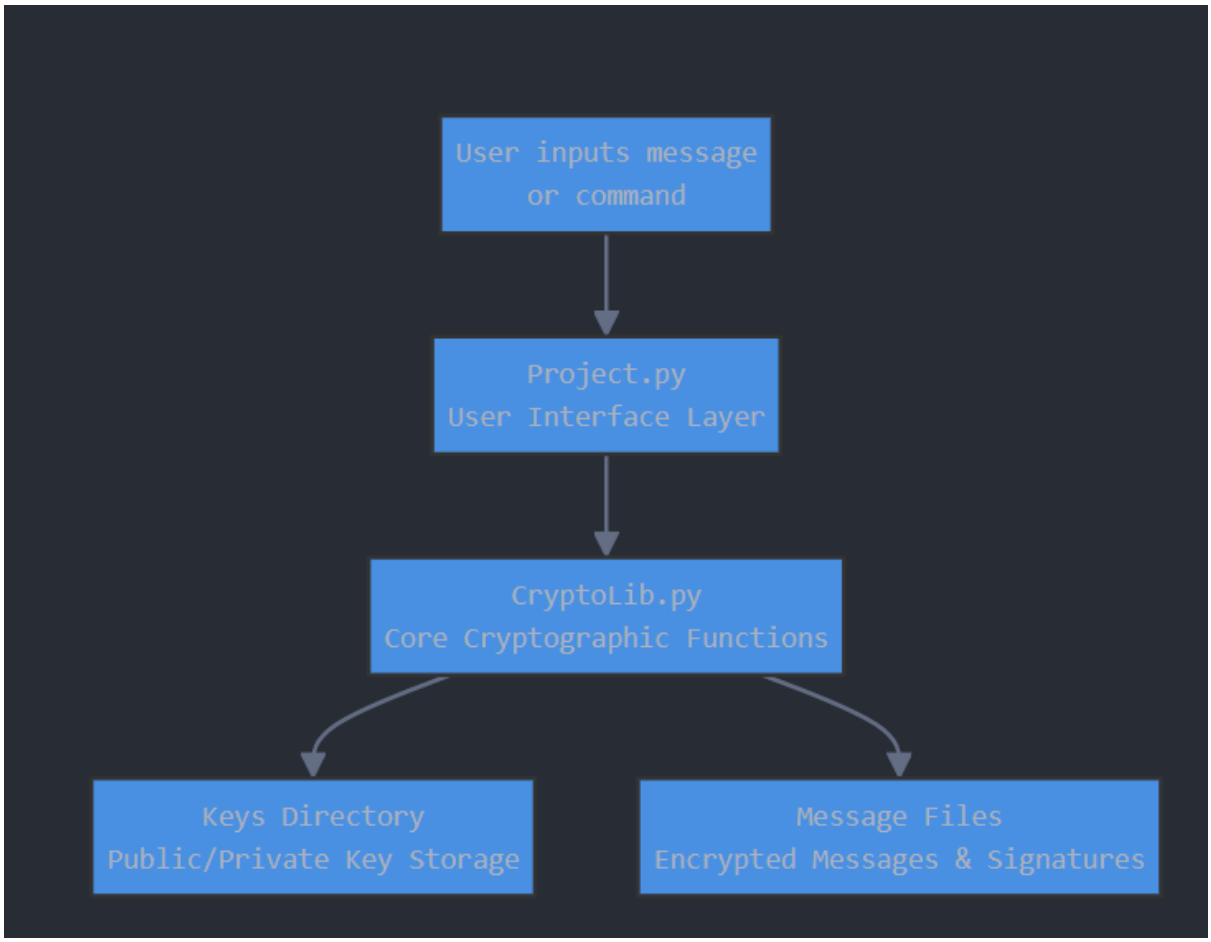


Figure 6.1: System Architecture Overview

The system consists of these key components:

- **Project.py**: Acts as the user interface layer, handling all user interactions and command processing
- **CryptoLib.py**: Contains all core cryptographic functions and utilities for encryption, decryption, and signature operations

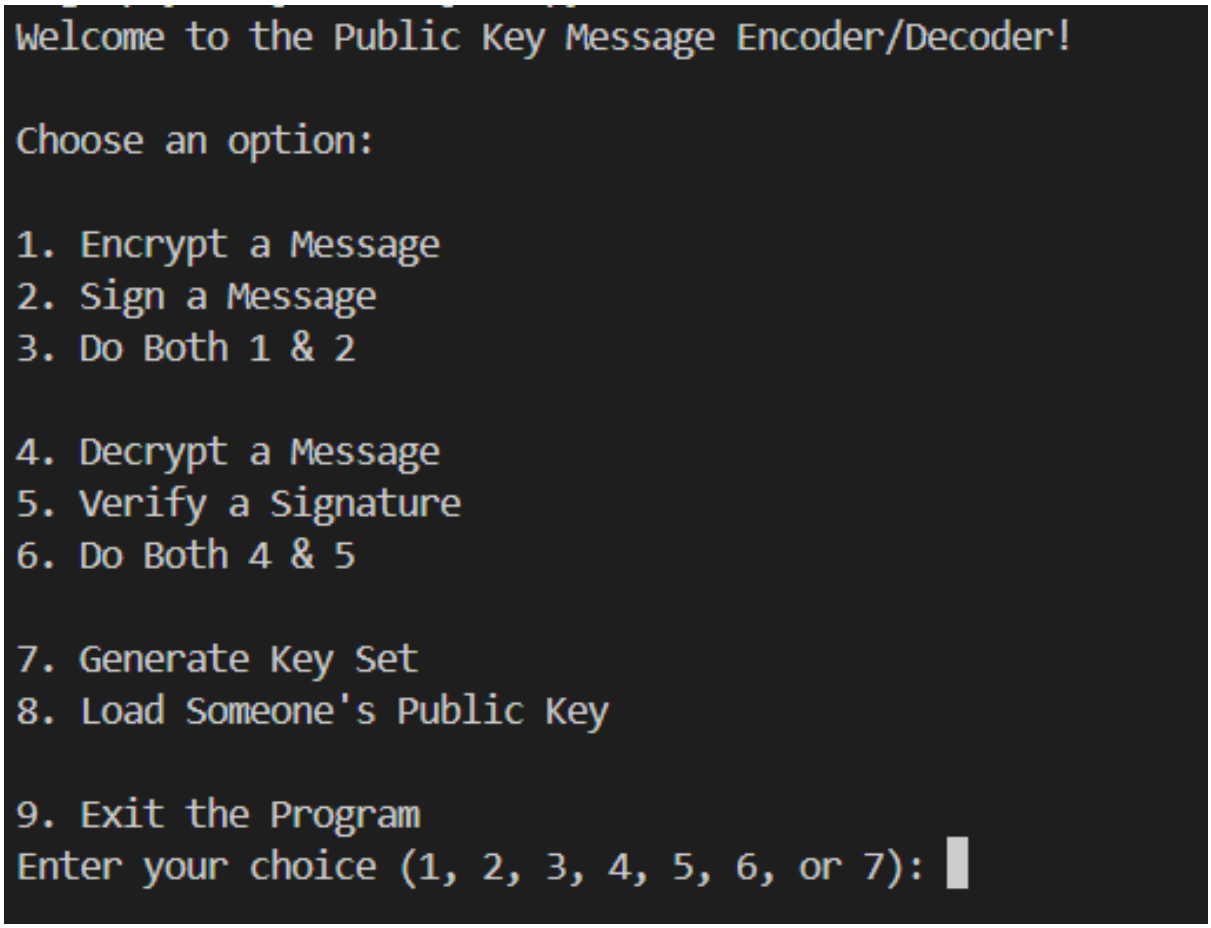
- **Keys Directory (/keys)**: Storage for Generated Private and Public Key Pairs
- **Message Files (message.txt)**: Holds the Encrypted Message.
- **Signature File (signature.txt)**: Holds the Sigtuare for a Given Message.

Chapter 7

Screenshots

7.1 All Screenshots

7.1.1 Screenshot 1



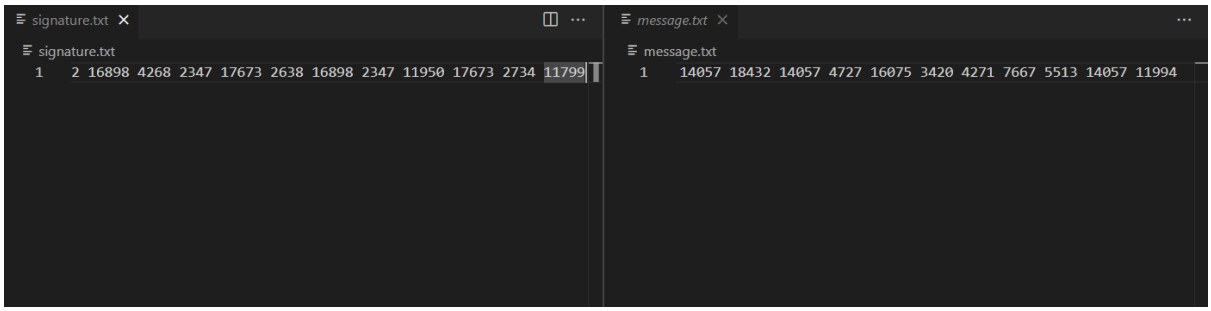
7.1.2 Screenshot 2

```
Enter your choice (1, 2, 3, 4, 5, 6, or 7): 7
Working, please wait....
New key set generated and saved in 'keys' directory:
Public Key (e,n): (5419, 19879)
Private Key (d,n): (6787, 19879)
You can now share your 'public key' with someone who wants to encrypt.
DO NOT SHARE YOUR PRIVATE KEY! THAT IS ONLY MEANT FOR DECRYPTION!
Enter your choice (1, 2, 3, 4, 5, 6, or 7):
```

7.1.3 Screenshot 3

```
Enter your choice (1, 2, 3, 4, 5, 6, or 7): 3
Enter message to encrypt and sign: Hello! This is a Test of the Project Created by Pranav Verma.
Encrypted message and signature saved to message.txt and signature.txt
Enter your choice (1, 2, 3, 4, 5, 6, or 7):
```

7.1.4 Screenshot 4



7.1.5 Screenshot 5

```
Enter your choice (1, 2, 3, 4, 5, 6, or 7): 6
Signature verified! Decrypted message: Hello! This is a Test of the Project Created by Pranav Verma.
Enter your choice (1, 2, 3, 4, 5, 6, or 7):
```

Chapter 8

Codes

8.1 Project.py

```
import CryptoLib as lb;
import os;
import sys;

print("Welcome to the Public Key Message Encoder/Decoder!")

print("\nChoose an option:")
print("")
print("1. Encrypt a Message")
print("2. Sign a Message")
print("3. Do Both 1 & 2")
print("")
print("4. Decrypt a Message")
print("5. Verify a Signature")
print("6. Do Both 4 & 5")
print("")
print("7. Generate Key Set")
print("8. Load Someone's Public Key")
print("")
print("9. Exit the Program")


while True:
    choice = input("Enter your choice (1, 2, 3, 4, 5, 6, or 7): "
   )

    # number 1
    if choice == "1":
        public_key = lb.load_key('keys/public.key')
        message = input("Enter the message to encrypt: ")
        encrypted_message = lb.encryptMessage(message, public_key
   )
        with open('message.txt', 'w') as f:
            f.write(encrypted_message)
        print("Encrypted message saved to message.txt")
```

```
 # number 2
 elif choice == "2":
     private_key = lb.load_key('keys/private.key')
     message = input("Enter the message to sign: ")
     signature = lb.sign_message(message, private_key)
     with open('signature.txt', 'w') as f:
         f.write(signature)
     print("Message signed and saved to signature.txt")
     print("Share both the message and signature.txt with the
recipient")

 # number 3
 elif choice == "3":
     recipient_public_key = lb.load_key('keys/public.key')
     sender_private_key = lb.load_key('keys/private.key')
     message = input("Enter message to encrypt and sign: ")
     encrypted_message, signature = lb.encrypt_and_sign(
message, recipient_public_key, sender_private_key)
     with open('message.txt', 'w') as f:
         f.write(encrypted_message)
     with open('signature.txt', 'w') as f:
         f.write(signature)
     print("Encrypted message and signature saved to message.
txt and signature.txt")

 # number 4
 elif choice == "4":
     private_key = lb.load_key('keys/private.key')
     filename = 'message.txt'
     if not os.path.exists(filename):
         print("File " + filename + " Not Found.")
         exit()
     with open(filename, 'r') as f:
         encrypted_message = f.read().strip()
     decrypted_message = lb.decrypt_message(encrypted_message,
 private_key)
     print("Decrypted Message:", decrypted_message)

 # number 5
 elif choice == "5":
     public_key = lb.load_key('keys/public.key')
     message = input("Enter the original message: ")
     filename = 'signature.txt'
     if not os.path.exists(filename):
         print("File " + filename + " Not Found.")
         exit()
     with open(filename, 'r') as f:
         signature = f.read().strip()
     if lb.verify_signature(message, signature, public_key):
         print("Signature is valid! Message is authentic.")
```

```
        else:
            print("Invalid signature! Message may have been
tampered with.")

    # number 6
    elif choice == "6":
        sender_public_key = lb.load_key('keys/public.key')
        recipient_private_key = lb.load_key('keys/private.key')
        if (not os.path.exists('signature.txt')):
            print("'signature.txt' Not Found.")
            print("Try Encrypting a Message First.")
            exit()
        if (not os.path.exists('message.txt')):
            print("'message.txt' Not Found.")
            print("Try Encrypting a Message First.")
            exit()
        with open('message.txt', 'r') as f:
            encrypted_message = f.read().strip()
        with open('signature.txt', 'r') as f:
            signature = f.read().strip()
        verified, result = lb.verify_and_decrypt(
encrypted_message, signature, sender_public_key,
recipient_private_key)
        if verified:
            print("Signature verified! Decrypted message:",
result)
        else:
            print("Error:", result)

    # number 7
    elif choice == "7":
        print("Working, please wait....")
        public_key, private_key = lb.generate_keys()
        print("New key set generated and saved in 'keys'
directory:")
        print("Public Key (e,n):", public_key)
        print("Private Key (d,n):", private_key)
        print("You can now share your 'public key' with someone
who wants to encrypt.")
        print("DO NOT SHARE YOUR PRIVATE KEY! THAT IS ONLY MEANT
FOR DECRYPTION!")

    # number 8
    elif choice == "8":
        print("Enter the public key in the format 'e,n'")
        print("Example: 6553,2341")
        key_text = input("Enter public key: ")
        if lb.saveKeyFromText(key_text):
            print("Public key successfully loaded and saved to
keys/public.key")
        else:
```

```
            print("Error: Invalid key format. Please use the
   format 'e,n' with numbers only")

    # number 9
    elif choice == "9":
        sys.exit(0)

    else:
        print("Please enter 1, 2, 3, 4, 5, 6, or 7.")
```

8.2 CryptoLib.py

```
# Crypto Library w/ all Functions.
# Made by Pranav Verma from XI Raman

import random;
import math;
import os;
import hashlib;

# standard gcd function
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def generate_keys():
    primes = []
    for i in range(100, 501):
        is_prime = True
        for j in range(2, int(math.sqrt(i)) + 1):
            if i % j == 0: #check for prime
                is_prime = False
                break
        if is_prime:
            primes.append(i)

    p = random.choice(primes) # random select prime 1
    q = random.choice(primes) # random select prime 2
    while p == q: # the two numbers should not be equal to the
   other
        q = random.choice(primes)

    n = p * q # calc. base
    phi = (p - 1) * (q - 1) #eulers totient function

    e = random.randint(2, phi - 1)
    while gcd(e, phi) != 1: # calc gcd
        e = random.randint(2, phi - 1)
```

```
    d = pow(e, -1, phi)

    if not os.path.exists('keys'):
        os.makedirs('keys')

    with open('keys/public.key', 'w') as f:
        f.write(f"{e},{n}")

    with open('keys/private.key', 'w') as f:
        f.write(f"{d},{n}")

    return ((e, n), (d, n))

def load_key(filename):
    if not os.path.exists(filename):
        print("Please generate keys first.")
        exit()
    with open(filename, 'r') as f:
        key = f.read().strip().split(',')
        return (int(key[0]), int(key[1]))

def saveKeyFromText(key_text):
    try:
        e, n = map(int, key_text.strip().split(','))
        if not os.path.exists('keys'):
            os.makedirs('keys')

        with open('keys/public.key', 'w') as f:
            f.write(f"{e},{n}")
        return True
    except:
        return False

def encryptMessage(message, public_key):
    e, n = public_key
    encrypted_message = []
    for char in message:
        encrypted_char = (ord(char) ** e) % n
        encrypted_message.append(str(encrypted_char))
    return ' '.join(encrypted_message)

def decrypt_message(encrypted_message, private_key):
    d, n = private_key
    encrypted_chars = encrypted_message.split()
    decrypted_message = ''.join([chr((int(char) ** d) % n) for
   char in encrypted_chars])
    return decrypted_message

def hash_message(message):
    return hashlib.sha256(message.encode()).hexdigest()
```

```
def sign_message(message, private_key):
    d, n = private_key
    message_hash = hash_message(message)
    signature = []
    for char in message_hash:
        # ok how did this actually work
        signed_char = pow(ord(char), d, n)
        signature.append(str(signed_char))
    return ' '.join(signature)

def verify_signature(message, signature, public_key):
    e, n = public_key
    # make the hash from original sig.
    signature_nums = signature.split()
    recovered_hash = ''
    for num in signature_nums:
        char = pow(int(num), e, n)
        recovered_hash += chr(char)
    # compare
    return recovered_hash == hash_message(message)

def encrypt_and_sign(message, recipient_public_key,
   sender_private_key):
    encrypted_message = encryptMessage(message,
   recipient_public_key)
    signature = sign_message(encrypted_message,
   sender_private_key)
    return encrypted_message, signature

def verify_and_decrypt(encrypted_message, signature,
   sender_public_key, recipient_private_key):
    # verify sig
    if not verify_signature(encrypted_message, signature,
   sender_public_key):
        return False, "Signature verification failed!"
    # decrpt the message.
    decrypted_message = decrypt_message(encrypted_message,
   recipient_private_key)
    return True, decrypted_message
```

Chapter 9

Bibliography

1. Python Documentation - `hashlib` module
2. Python Documentation - `random` module
3. GeeksForGeeks - `Eulers Totient Function`
4. JavatPoint - `RSA Formula`